Codezero代码目录和编译流程

柯杰伟

May 20, 2014

1 运行环境搭建

Codezero Reference Manual第一部分给出了详细的运行环境搭建步骤。但其中的个别下载链接不可用，同时安装过程中有些错误在文档中没有列举，故编写了以下脚本init-build-env.sh，便于自动化配置运行环境。

init-build-env.sh的内容如下：

```
#!/bin/bash
# written by kjw, 2014.5.5

echo "1. Creating the directory structure for Codezero sources and tools"
sudo mkdir /opt

echo "2. Installing Git"
sudo apt-get install git-core

echo "3. Installing SCons"
sudo apt-get install scons

echo "4. Installing the GCC cross-compiler"
sudo apt-get install curl

if !(command -v /opt/arm-2009q3/bin/arm-none-eabi-gcc > /dev/null 2 >& 1);then
    sudo curl -L "https://sourcery.mentor.com/public/gnu_toolchain/arm-none-eabi/arm-2009
fi

if !(command -v /opt/arm-2009q3/bin/arm-none-linux-gnueabi-gcc > /dev/null 2 >& 1);then
    sudo curl -L "https://smp-on-qemu.googlecode.com/files/arm-2009q3-67-arm-none-linux-g
fi

if !(grep -q "arm-2009q3" ~/.bashrc); then
    echo -e "PATH=$PATH:/opt/arm-2009q3/bin\nexport PATH" >> ~/.bashrc
    source ~/.bashrc
fi

echo "5. Building Codezero sources"
cd ~
git clone https://github.com/jserv/codezero.git jserv-codezero
cd jserv-codezero
chmod a+x build.py
./build.py --configure
./build.py

echo "6. Installing QEMU"
if !(command -v qemu-system-arm > /dev/null 2 >& 1);then
    sudo curl -L "http://wiki.qemu-project.org/download/qemu-0.13.0.tar.gz" | gzip -d --s
```

```
    cd /opt/qemu-0.13.0
    sudo ./configure
    sudo make && make install

# Q1: undefined reference to symbol timer_settime@@GLIBC_2.3.3
# Edit Makefile.target in your qemu directory, find LIBS+=-lz, add LIBS+=-lrt beneath thi

# Q2: error: field info has incomplete type
# struct siginfo info; => siginfo_t info;
fi

#echo "7. Installing Insight"
#sudo mkdir /opt/insight
#cd /opt
#echo "[enter 'anoncvs' as the password]"
#cvs -z9 -d :pserver:anoncvs@sourceware.org:/cvs/src login
#sudo cvs -z9 -d :pserver:anoncvs@sourceware.org:/cvs/src co -r gdb_6_8-branch insight
#https://sourceware.org/insight/faq.php

echo "8.GDB configuration"
cd ~/jserv-codezero
cp ./tools/gdbinit ~/.gdbinit

echo "9.Running Codezero"
cd ~/jserv-codezero
echo -e "cd build\nqemu-system-arm -s -kernel final.elf -m 128 -M versatilepb -nographic;
chmod a+x ./tools/run-my-qemu
./tools/run-my-qemu
%
```

2 Codezero代码目录

2.1 loader，5428行

该目录包含了ELF库和boot loader的源码。boot loader用于遍历加载内核和container镜像。
Codezero构建结束会生成final.elf映像，其中打包了kernel和container，以及loader程序。
loader/libs目录包含了C和ELF两个库。都是用于加载的。

2.2 tools

包含了一些工具。tools/tagsgen用于对内核和其他项目产生ctags和cscopt索引数据。tools/ run-qemu-insight用于加载运行final.elf时QEMU和Insight/GDB的初始化。

2.3 src 14139行,include 6215行

包含了Codezero微内核的源码。

2.4 conts

包含了所有用户空间代码，即每个container运行的代码。
conts/baremetal包含了Hello World,用户程序模板等示例程序。
conts/linux用于在Codezero上运行虚拟化的Linux。这里没有该目录，需要额外下载配置。
conts/uboot用于在Codezero上运行虚拟化的u-boot实例。这里没有该目录，需要额外下载配置。
conts/userlibs包含了用户空间库函数，libl4/libc/libmem/libdev。
conts/posix

3 Codezero编译流程

4 启动流程分析

4.1 分析情景

Codezero是基于L4体系的微内核，可以虚拟操作系统，也可以在其上面直接虚拟运行App，本文分析基于ARM926体系的helloworld应用进行分析。

4.2 ARM926介绍

ARM926属于ARMv5架构，具有增强的32-bit RISC CPU，以及灵活的指令和数据Cache，TCM（tightly coupled memory）接口，以及MMU。并且ARM926提供分离的指令和数据AMBA、AHB接口以适应基于AHB的多层系统设计。

4.3 系统配置

配置文件采用jserv-codezero/scripts/config/cml/examples/helloworld/config.cml，配置、编译方法为：./build.py -f ./scripts/config/cml/examples/helloworld/config.cml。

在此配置文件里，主要的参数如：

CONFIG_CONT0_PHYSMEM_REGIONS=1

CONFIG_CONT0_PHYS0_START=0x100000

CONFIG_CONT0_PHYS0_END=0xe00000

在此配置了CONT0的起始物理地址，终止物理地址，但是在Qemu-里执行的时候，发现系统启动的物理地址是从0xe00000开始的。查看生成的final.elf也发现入口地址为0xe00000;

```
$ arm-none-linux-gnueabi-readelf -a ./build/final.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0xe00000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1190336 (bytes into file)
  Flags:                             0x5000002, has entry point, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         19
  Section header string table index: 16
```

4.4 启动过程

4.4.1 概述

启动过程主要包括系统上电之后，首先处理器跳转到指定的物理地址去加载第一条指令执行，在这里就是loader，之后处理器将控制权交给loader，loader在完成必要的初始化之后，加载内核的第一个汇编文件接着执行，在这里是Head.S，然后内核继续初始化硬件，控制权交由内核处理，内核在完成初始化，系统配置之后，加载container，然后使能MMU，进入虚拟地址空间进行虚拟应用的执行，至此启动过程结束。

4.4.2 Loader

处理器上电，首先执行的是loader，类似于嵌入式linux里的uboot，不过codezero里的loader要简单的多，代码量也少很多。下面以程序运行的顺序介绍用到的函数:

Crt0.S（jserv-codezero/loader/libs/c/crt/sys-baremetal/arch-arm）

该函数很简单，汇编编写:

```
                .section .text
                .code 32
                .global _start;
                .align;
_start:
                ldr    sp,
                bl     platform_init
                bl     main
1:      .word     _stack_top
                .bss
                .align
_stack:
                .space     1024
_stack_top:
```

完成的功能就是加载PC指针到SP，然后调用platform_init进行必要的硬件初始化，然后调用loader的man函数，进行codezero kernel以及container的加载执行。

反汇编看的很清楚的程序起始跳转是在0xe00000处:

```
0x00e00000 <+0>:     ldr     sp, [pc, #4]     ; 0xe <_start+12>
0x00e00004 <+4>:     bl      0xe10 <platform_init>
0x00e00008 <+8>:     bl      0xe0 <main>
0x00e <+12>:     rsceq   r3, pc, r12, asr #19
```

platform_init（jserv-codezero/src/platform/pb926/platform.c）

该函数完成必要的硬件初始化，如下:

```
void platform_init(void)
{
        init_platform_console();
        init_platform_timer();
        init_platform_irq_controller();
        init_platform_devices();
}
```

main（jserv-codezero/loader/mail.c） 函数的主要作用为加载kernel image、container image，然后通过函数arch_start_kernel引导内核运行，在arch_start_kernel里用到kernel的kernel_entry，该参数在gdbinit里配置，如果没有配置其存在于0x8000地指处。

从kernel_entry开始，loader的功能全部完成，系统进入到codezero的kernel运行。

4.4.3 kernel

Kernel的入口函数为head.S，codezero的内核文件构成参照了标准linux内核的文件结构。

Head.S(jserv-codezero/src/arch/arm) 该函数是kernel的入口点，在该程序里，首先关闭MMU、指令、数据Cache、写缓冲等，然后定义必要的向量表，最后跳转到start_kernel继续执行。

start_kernel（jserv-codezero/src/glue/arm） 继续进行初始化，在start_vm函数里打开MMU，系统进入虚拟地址执行，最后在jump函数里加载虚拟应用进行执行。

Jump函数由汇编完成，函数如下:

```
void jump(struct ktcb *task)
{
        __asm__ __volatile__ (
                "mov␣␣␣␣␣␣lr,␣␣␣␣%0\n"     /* Load pointer to context area */
                "ldr␣r0,␣␣␣[lr]\n"      /* Load spsr value to r0 */
```

```
                "msr␣␣␣␣␣␣␣␣spsr,␣␣␣␣␣␣␣␣r0\n"        /* Set SPSR as ARM_MODE_USR */
                "add␣␣␣␣␣␣␣sp,␣lr,␣%1\n" /* Reset SVC stack */
                "sub␣␣␣␣␣␣␣␣sp,␣sp,␣%2\n"          /* Align to stack alignment */
                "ldmib␣␣␣lr,␣{r0-r14}^\n" /* Load all USR registers */
                "nop␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"    /* Spec says dont touch banked registers
                                         * right after LDM {no-pc}^ for one instruction */
                "add␣␣␣␣␣␣␣lr,␣lr,␣#64\n"    /* Manually move to PC location. */
                "ldr␣lr,␣␣␣␣[lr]\n"       /* Load the PC_USR to LR */
                "movs␣␣␣␣pc,␣lr\n"         /* Jump to userspace, also switching SPSR/CPSR */
                :
                : "r" (task), "r" (PAGE_SIZE), "r" (STACK_ALIGNMENT)
        );
}
```

4.4.4 container

__container_init（jserv-codezero/conts/hello_world0）

该函数作为app的入口，首先初始化L4体系__l4_init();然后调用app的main函数。

main（jserv-codezero/conts/hello_world0）

Main函数为app的实际入口函数，应用程序的编写在这里实现，在helloworld demo里调用了hello.c文件里的print_hello_world函数。